Cleaning Relations using Knowledge Bases

Shuang Hao[†] Nan Tang[‡] Guoliang Li[†] Jian Li[†]
[†]Department of Computer Science, Tsinghua University [‡]Qatar Computing Research Institute, HBKU
{haos13, liguoliang, lijian83}@tsinghua.edu.cn, ntang@hbku.edu.qa

***Abstract*—We study the data cleaning problem of detecting and repairing wrong relational data, as well as marking correct data, using well curated knowledge bases (KBs). We propose detective rules (DRs), a new type of data cleaning rules that can make actionable decisions on relational data, by building connections between a relation and a KB. The main invention is that, a DR simultaneously models two opposite semantics of a relation using types and relationships in a KB: the *positive semantics* that explains how attribute values are linked to each other in correct tuples, and the *negative semantics* that indicates how wrong attribute values are connected to other correct attribute values within the same tuples. Naturally, a DR can mark correct values in a tuple if it matches the *positive semantics*. Meanwhile, a DR can detect/repair an error if it matches the *negative semantics*. We study fundamental problems associated with DRs, *e.g.,* rule generation and rule consistency. We present efficient algorithms to apply DRs to clean a relation, based on rule order selection and inverted indexes. Extensive experiments, using both real-world and synthetic datasets, verify the effectiveness and efficiency of applying DRs in practice.**

I. Introduction

In industries, all data analysts report that they spend more than 80% of time doing the "grunt work" of data cleaning before data analytics. There are many studies in cleaning data using integrity constraints (ICs) [4]–[6], [25], [27]. ICs are good at capturing errors. However, the serious drawback of ICs is that they cannot precisely tell which value is wrong. Take functional dependencies (FDs) for example. Consider an FD country → capital over the relation $R(\mathsf{country}, \mathsf{capital})$, and two tuples t_1(*China, Beijing*) and t_2(*China, Shanghai*). The FD can identify the existence of errors in t_1 and t_2, but cannot tell which value is wrong. All FD-based repairing algorithms use some heuristics to guess the wrong value, $t_1[\mathsf{country}]$, $t_1[\mathsf{capital}]$, $t_2[\mathsf{country}]$, or $t_2[\mathsf{capital}]$, and then repair it.

In contrast, rule-based data repairing explicitly tells how to repair an error. For instance, fixing rules [28] can specify that for each tuple, if its country is *China* and its capital is *Shanghai*, then *Shanghai* is wrong and should be changed to *Beijing*. Other rule-based approaches such as editing rules [16] and Sherlock rules [20] use tabular master data, to collect evidence from external reliable data sources.

Arguably, both IC- and rule-based methods can make mistakes when repairing data. However, IC-based tools are more like *black-boxes* while rule-based methods are *white-boxes*. When some mistakes made by the tools are identified, the latter is more interpretable about what happened. Not surprisedly, in industries, rule-based repairing methods are widely adopted, *e.g.,* ETL rules, but IC-based tools are rarely employed.

Nowadays, we are witnessing an increased availability of well curated KBs such as Yago [19] and DBpedia [23]. Also, large companies maintain their own KBs *e.g.,* Warlmart [11], Google and Microsoft. In order to take advantage of these KBs, we extend prior rule-based cleaning methodologies [16], [20] to clean relations by collecting evidence from KBs. The core of our proposal is a new type of data cleaning rules that build the connections between relations and KBs. They are rule-based, such that the actions of how to clean the data are baked in the rules, which rely on neither other heuristic solutions as those for ICs [4]–[6], nor the domain experts [7], [24].

Motivating Examples. Perhaps the best way of understanding our proposal is by examples.

Example 1: [Relation.] Consider a database D of Nobel laureate records, specified by the following schema:

$$\mathsf{Nobel}\ (\mathsf{Name}, \mathsf{DOB}, \mathsf{Country}, \mathsf{Prize}, \mathsf{Institution}, \mathsf{City}),$$

where a Nobel tuple specifies a Nobel laureate in Chemistry, identified by Name, together with its DOB, Country, Prize, Institution and City of the institute. Table I shows four tuples. All errors are highlighted and their correct values are given between brackets. For instance, consider r_1 about *Avram Hershko*, a Hungarian-born Israeli biochemist. The value $r_1[\mathsf{City}]$ = *Karcag* is an error, which is the city he was born in, whose correct value is *Haifa* where he works in. □

Next we discuss, based on the available evidence from KBs, how to make judgement on the correctness of a relation.

Example 2: [Knowledge Base.] Consider an excerpt of a KB Yago [19], as depicted in Figure 1. Here, a solid rectangle represents an *entity*, *e.g.,* u_1 to u_7 , a dotted rectangle indicates a *class e.g.,* country and city, a shaded rectangle is a *literal e.g.,* u_8, a labeled and directed edge shows the *relationship* between entities or the *property* from an entity to a literal, and a dotted edge associates an entity with a class.

Consider tuple r_1 in Table I and the sample KB in Figure 1, both about *Avram Hershko*. It is easy to see that most values of r_1 appear in Figure 1. Based on different bindings of relationships, we can have the following three actions for r_1.

(i) *Proof Positive.* Based on the evidence in Figure 1, we may judge that $r_1[\mathsf{Name}, \mathsf{DOB}, \mathsf{Country}, \mathsf{Institution}]$ are correct.

(ii) *Proof Negative.* If we know that, $r_1[\mathsf{City}]$ should be the city that he works in, and we find from Figure 1 that (a) *Karcag* is the city he was born in; (b) *Haifa* is the city he works in, via the links *Avram Hershko* worksAt *Israel Institute of Technology* that in turn locatedIn *Haifa*; and (c) *Karcag* is different from

Guoliang Li is the Corresponding Author.

	Name	DOB	Country	Prize	Institution	City
r_1	Avram Hershko	1937-12-31	Israel	Albert Lasker Award for Medicine (Nobel Prize in Chemistry)	Israel Institute of Technology	Karcag (Haifa)
r_2	Marie Curie	1867-11-07	France	Nobel Prize in Chemistry	Paster Institute Pasteur Institute	Paris
r_3	Roald Hoffmann	1937-07-18	Ukraine (United States)	National Medal of Science (Nobel Prize in Chemistry)	Cornell University	Ithaca
r_4	Melvin Calvin	1911-04-08	United States	Nobel Prize in Chemistry	University of Minnesota (UC Berkeley)	St. Paul (Berkeley)

TABLE I. DATABASE D: NOBEL LAUREATES IN CHEMISTRY

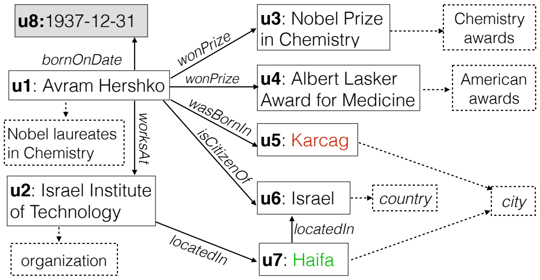


Fig. 1. Excerpt of laureates knowledge bases

Haifa, we can judge that r_1[City] is wrong. The way to identity the error in r_1[Prize] is similar, if we know this column should be Chemistry awards rather than American awards.

(*iii*) *Correction.* Following (*ii*), we can draw the value *Haifa* from the KB to update r_1[City] from *Karcag* to *Haifa*. □

Challenges. Example 2 shows that we can make different judgements, based on various evidence from KB. Nevertheless, effectively employing reliable KBs faces several challenges.

(i) *Semantic Connections between Relations and* KB*s.* In order to collect evidence from KBs and judge on the relations at hand, it requires to build graphical (semantic) connections between tables and KBs.

(ii) *Ambiguity of Repairing.* A typical ambiguity raised by IC-based approaches is that they cannot tell precisely which attribute value is to be repaired. Hence, we need to explicitly specify which attribute is wrong and how to repair, a departure from traditional ICs that only detect errors.

(iii) *Efficiency and Scalability.* Repairing a relation using multiple rules by collecting evidence from a large KB (a graph) is a costly task, which requires efficient and scalable solutions.

Contributions. Our main contribution is to propose a new type of rules to deterministically tell how to repair relations using KBs. We summarize our contributions below.

(1) We formally define detective rule (DR), to address challenges (i) and (ii). A DR simultaneously models two opposite semantics of a relation using types and relationships in a KB (Section II): the *positive semantics* that explains how attribute values should be linked to each other in correct tuples, and the *negative semantics* that indicates how wrong attribute values are connected to other correct attribute values within the same tuples. Naturally, a DR can mark correct values in a tuple if it matches the *positive semantics*. Meanwhile, a DR can detect/repair an error if it matches the *negative semantics*.

(2) We study several fundamental problems of using DRs (Section III). Specifically, we describe the generation of DRs. We discuss the semantics of applying a set of DRs. Also, we study some theoretical problems such as consistency analysis associated with DRs.

(3) We devise efficient algorithms for cleaning a relation, given a set of consistent DRs, by smartly selecting the right order to apply DRs and by using various indexes such as rule indexes and signature-based indexes (Section IV). This is to cope with challenge (iii).

(4) We experimentally verify the effectiveness and efficiency of the proposed algorithms (Section V). We find that algorithms with DRs can repair and mark data with high accuracy. In addition, they scale well with the number of DRs.

Related Work. We categorize related work as follows.

Contraint-based Data Cleaning. IC-based heuristic data cleaning methods have been widely studied [4], [8], [9], [12] for the problem introduced in [2]: repairing is to find another database that is consistent and minimally differs from the original database. However, the consistency may not be an ideal objective, since the ground truth database is consistent, but not vice versa. In contrast, rule-based data cleaning is typically more conservative and reliable, since it does not use heuristics. DRs only mark data as correct or wrong, and repair errors when the evidence is sufficient.

Rule-based Data Cleaning. Different methods exist in the literature regarding rule-based data cleaning: editing rules [16], fixing rules [28], and Sherlock rules [20]. Editing rules [16] use relational master data and interact with users for trusted repairing. Fixing rules [28] encode constant values in the rules for automated cleaning. Closer to this work is Sherlock rules [20] that automatically annotate and repair data. Along the same line with them, DRs are the research effort to leverage KBs for data cleaning. The new challenges of using KBs are remarked earlier in this section.

Table Understanding using KBs. Table understanding, including identifying column types and the relationship between columns using KBs, has been addressed by several techniques such as those in [10], [22], [26]. In fact, these techniques are friends, instead competitors, of DRs. We will show how they can help to discover DRs (Section III-A).

KB Powered Data Cleaning. KATARA [7] is a KB and crowd powered data cleaning system that identifies correct and incorrect data. The table patterns [7] introduced by KATARA are a way of explaining table semantics in a holistic way. However, (1) KATARA cannot detect errors automatically: Whenever a mismatch happens between a tuple and a KB *w.r.t.* a table pattern, KATARA will ask the crowd workers to identify that such a mismatch is caused by an error in the tuple, or an

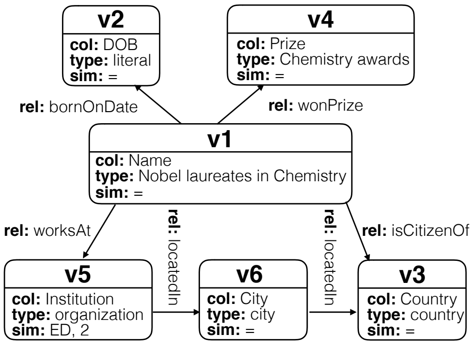


Fig. 2. A sample schema-level matching graph

incompleteness of the KB; and (2) KATARA cannot repair errors automatically: When an error, such as (China, Shanghai) for relation (country, capital), is identified by users, KATARA cannot tell which value is wrong. One main difference between DRs and KATARA is that DRs can precisely tell which attribute of a tuple is wrong.

User Guided Data Cleaning. Several approaches [16], [18], [24], [29] have been proposed to involve experts as first-class citizen. Involving users is certainly valid and useful for specific applications. DRs are our attempt to relieve users from the tedious and iterative data cleaning process.

II. DETECTIVE RULES

We first introduce notations for knowledge bases (KBs) (Section II-A). We then present the basic concepts of building connections between relations and KBs (Section II-B). We close this section by defining detective rules (Section II-C).

A. Knowledge Bases

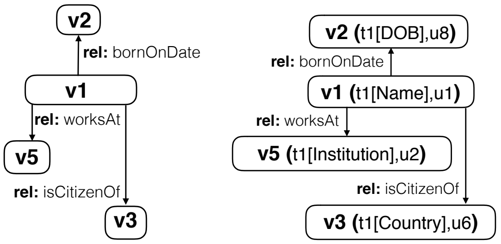
We consider KBs as RDF-based data, defined using Resource Description Framework Schema (RDFS).

Classes, Instances, Literals, Relationships, and Properties. A *class* represents the concept of a set of entities, *e.g.,* country. An *instance* represents an entity, *e.g., Israel*, which belongs to a class, *e.g.,* type (*Israel*) = country. A *literal* is a string, a date, or a number. For example, the birth date of *Avram Hershko* is *1937-12-31*, which is a literal. A *relationship* is a binary predicate that represents a connection between two instances. For instance, isCitizenOf(*Avram Hershko, Israel*) indicates that *Avram Hershko* is a citizen of *Israel*, where isCitizenOf is a relationship defined in a KB. An instance can have some properties, *e.g.,* bornOnDate. A *property* is a binary predicate that connects an instance and a literal.

Let $\mathbf{I}$ be a set of instances, $\mathbf{L}$ a set of literals, $\mathbf{C}$ a set of classes, $\mathbf{R}$ a set of relationships, and $\mathbf{P}$ a set of properties.

RDF Graphs. An *RDF* dataset is a set of triples $\{(s, p, o)\}$, where s is an instance in $\mathbf{I}$, p is a relationship in $\mathbf{R}$ or a property in $\mathbf{P}$, o is an object in $\mathbf{I} \cup \mathbf{L}$ (*i.e.,* either an instance or a literal). We model the set of triples $\{(s, p, o)\}$ as a directed graph. Each vertex v is either an s or an o from the given triples. Each directed edge $e : (s, o)$ corresponds to a triple (s, p, o), with p as the edge label denoted by $\mathsf{rel}(e) = p$.

Please refer to Figure 1 as a sample RDF graph, which describes an excerpt of Yago describing *Avram Hershko*.



(a) schema-level (b) instance-level

Fig. 3. A sample schema/instance-level matching graph

B. Schema- and Instance-Level Matching Graphs

Given a table D of schema R, and a KB K, next we discuss how to build connections between them, a necessary step to collect evidence from K for D. Generally speaking, we need schema-level matching graphs to explain the schema of D using K, and instance-level matching graphs to find values in K that correspond to tuples in D.

Schema-Level Matching Graphs. A *schema-level matching graph* is a graph $\mathbf{G}^{\mathbf{S}}(\mathbf{V}^{\mathbf{S}}, \mathbf{E}^{\mathbf{S}})$, where:

1) each vertex $u \in \mathbf{V}^{\mathbf{S}}$ specifies a *match* between a column in D and a type in K. It contains three labels:
   a) $\mathsf{col}(u)$: the corresponding column in D;
   b) $\mathsf{type}(u)$: a type in K - either a class or a literal;
   c) $\mathsf{sim}(u)$: a similarity based matching operation.
2) for two different nodes $u, v \in \mathbf{V}^{\mathbf{S}}$, $\mathsf{col}(u) \neq \mathsf{col}(v)$.
3) each directed edge $e : (u, v) \in \mathbf{E}^{\mathbf{S}}$ has one label $\mathsf{rel}(e)$, which is a *relationship* or *property* in K, indicating how $\mathsf{col}(u)$ and $\mathsf{col}(v)$ are semantically linked.

An important issue is to define the matching operation $\mathsf{sim}(u)$ (the above 1(c)) between a column in a relation and a class in a KB, which will be used later to decide whether two values match. We can utilize similarity functions, *e.g.,* Jaccard, Cosine or edit distance. For example, if string equality "=" is used, a cell value in column $\mathsf{col}(u)$ and an instance in K with class $\mathsf{type}(u)$ refer to the same entity if they have identical value. If "ED, 2" is used, a cell value in column $\mathsf{col}(u)$ and an instance in K with class $\mathsf{type}(u)$ refer to the same entity if their edit distance[1] is within 2. Without loss of generality, we take string equality and edit distance as examples.

Example 3: [Schema-level matching graph.] A sample schema-level matching graph for the Nobel table in Table I is given in Figure 2. Node v_1 shows that column Name corresponds to the class Nobel laureates in Chemistry in the KB. To match a value in column Name and a value in the KB with type Nobel laureates in Chemistry, string equality "=" is used. "ED, 2" is used in node v_5 to tolerate the errors between r[Institution] and an instance of organization in KB. The directed edge (v_1, v_2) labelled bornOnDate explains the relationship between columns Name and DOB. □

Be default, we assume that a schema-level matching graph is connected. Naturally, any induced subgraph of a schema-level matching graph is also a schema-level matching graph. In

[1] Edit distance of two instances is the minimum number of edit transformations from one to the other, where the edit operations include insertion, deletion and substitution. For example ED(Chemistry, Chamstry) = 2.

other words, a schema-level matching graph is not necessarily a global understanding of the table (see Figure 2). In contrast, it is a local interpretation about how partial attributes of a table are semantically linked, *e.g.,* Figure 3(a). Please refer to Figure 2 for the corresponding node labels.

Construction. The schema-level matching graph is essentially the table semantics interpreted by a KB, which has been widely studied. A basic idea is to connect a column to a type by matching the cell-value set of the column to the entity set of the type and we can use existing tools [7], [10], [22] to build the schema-level matching graph.

Instance-Level Matching Graph. An *instance-level matching graph*, denoted by $\mathbf{G^I}(\mathbf{V^I}, \mathbf{E^I})$, is an instantiation of a schema-level matching graph *w.r.t.* one tuple t in the relational table and some instances from the KB. More formally speaking:

1) For each node $u_i \in \mathbf{V^I}$, there is an instance x_i from the KB such that the types match *i.e.,* $\mathsf{type}(u_i) = \mathsf{type}(x_i)$; and the values match *i.e.,* $t[\mathsf{col}(u_i)]$ and x_i are similar based on the similarity function $\mathsf{sim}(u_i)$.
2) For each edge $(u_i, u_j) \in \mathbf{E^I}$, the correspondingly matched KB instances x_i, x_j satisfy $\mathsf{rel}(u_i, u_j) = \mathsf{rel}(x_i, x_j)$, *i.e.,* the two instance x_i and x_j in the KB have the relationship required by the schema-level matching graph.

Example 4: [Instance-level matching graph.] Consider the small schema-level matching graph shown in Figure 3(a). One instance-level matching graph for tuple r_1 in Table I is given in Figure 3(b), where the types and relationships of the nodes u_1, u_2, u_6, u_8 can be verified from the KB in Figure 1. □

Limitations. Indeed, *matching operation* is the core of data cleaning, since one always needs to link different real-world entities. Historically, many matching operators have been studied, *e.g.,* matching dependencies [15] for two tables, keys for graphs [13] defined on one graph, and Swoosh [3] for matching two generic objects for entity resolution. When there is a match, one common usage is to say something is correct. The main limitation for detecting errors is that, when there is a mismatch, it cannot tell that something is wrong.

Opportunities. If we define some matching operations to capture negative semantics, intuitively, the errors can be detected. This observation reveals the opportunity to define new methods to match (*i.e.,* detect) data errors. For instance, in Section I Example 2 Case (ii), if we know that (1) the column City in the table is where he works in; (2) the current value t_1[City] is *Karcag*; (3) he works in *Haifa*, derived from the KB; and (4) the two values *Karcag* and *Haifa* are different, we may decide that *Karcag* is an error.

C. Detective Rules

The broad intuition of our proposal is that, for a column, if we can simultaneously capture both the *positive semantics* of what correct values should look like, and the *negative semantics* of what wrong values commonly behave, we can detect and repair errors.

Consider column City in Table I. We can discover one schema-level matching graph to capture the semantics *lives at*. Similarly, we can find another one to capture the semantics *born in*. If the user enforces City to have the *lives at* semantics, and we find that the value in the table maps to the *born in* semantics, we know how to repair. Note that some semantics can be captured by a directed edge *e.g.,* wasBornIn in Figure 1 for the *born in* semantics, while some other semantics needs to be captured by more than one edge *e.g.,* putting worksAt and locatedIn in Figure 1 together for the *lives at* semantics.

Let $\mathbf{G^S_1}(\mathbf{V^S_1}, \mathbf{E^S_1})$ and $\mathbf{G^S_2}(\mathbf{V^S_2}, \mathbf{E^S_2})$ be two schema-level matching graphs that exist a node $p \in \mathbf{V^S_1}$ and a node $n \in \mathbf{V^S_2}$ such that (i) $\mathsf{col}(p) = \mathsf{col}(n)$ and (ii) the subgraphs $\mathbf{G^S_1}\backslash\{p\}$ and $\mathbf{G^S_2}\backslash\{n\}$ are isomorphic, where $\mathbf{G^S_1}\backslash\{p\}$ (resp. $\mathbf{G^S_2}\backslash\{n\}$) is a subgraph of $\mathbf{G^S_1}$ (resp. $\mathbf{G^S_2}$) by removing node p (resp. n) and associated edges. Obviously, both graphs are defined over the same set of columns in the relation: $\mathbf{G^S_1}$ is to capture their positive semantics and $\mathbf{G^S_2}$ is for the negative semantics of values in column $\mathsf{col}(n)$.

Detective Rules. A *detective rule* is a graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$ that merges the above two graphs $\mathbf{G^S_1}$ and $\mathbf{G^S_2}$. Let $\mathbf{V_e} = \mathbf{G^S_1}\backslash\{p\} = \mathbf{G^S_2}\backslash\{n\}$. The node set in DR is $\mathbf{V} = \mathbf{V_e} \cup \{p, n\}$. The edges $\mathbf{E}$ are all the edges carried over from the above two graphs. Note that $\mathsf{col}(p) = \mathsf{col}(n)$. We call p the *positive node*, n the *negative node*, and $\mathbf{V_e}$ the evidence nodes.

Semantics. Let $\mathsf{col}(\mathbf{V_e})$ be the columns corresponding to the evidence nodes $\mathbf{V_e}$ of a DR, and $\mathsf{col}(p) = \mathsf{col}(n)$. Let $|\mathbf{V_e}|$ be the cardinality of the set $\mathbf{V_e}$. Consider a tuple t over relation R and a KB K:

(1) Proof Positive. If there is an instance-level matching graph between t and $|\mathbf{V_e}|+1$ instances in K *w.r.t.* the nodes $\mathbf{V_e} \cup \{p\}$, *i.e.,* the positive semantics is captured, we say that the attribute values of $t[\mathsf{col}(\mathbf{V_e}) \cup \mathsf{col}(p)]$ are correct.

(2) Proof Negative. If there is an instance-level matching graph between t and $|\mathbf{V_e}|+1$ instances in K *w.r.t.* the nodes $\mathbf{V_e} \cup \{n\}$, we say that the attribute values of $t[\mathsf{col}(\mathbf{V_e})]$ are correct, but the value $t[\mathsf{col}(n)]$ is potentially wrong, *i.e.,* the negative semantics is captured. In addition, if we can find another instance x in K such that if we replace $t[\mathsf{col}(n)]$ by x, we can find the case of proof positive as in (1). At this point, we confirm that $t[\mathsf{col}(n)]$ is wrong.

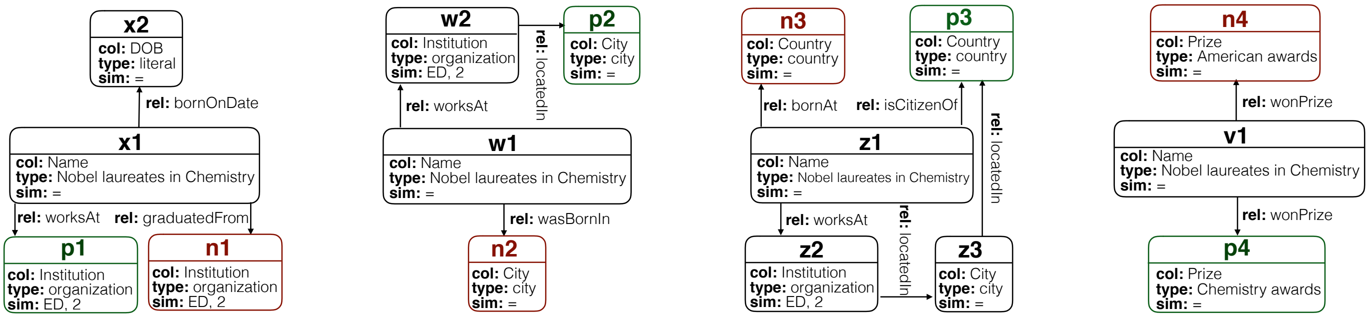
(3) Correction. Following the above case (2), we know the correct value for $t[\mathsf{col}(n)]$ is the new instance x from K.

Intuitively, a DR specifies how to judge a set of attribute values of a tuple is correct (the above (1)), how to find one wrong attribute value of the tuple (the above (2)), and how to repair the identified error (the above (3)).

Example 5: [Detective rules.] Figure 4 shows four DRs. We discuss their semantics *w.r.t.* r_1 in Table I and KB in Figure 1.

(1) Proof Positive. Consider rule φ_1. We can find that u_1 in Figure 1 matches r_1[Name] *w.r.t.* node x_1 in φ_1; u_8 in Figure 1 matches r_1[DOB] *w.r.t.* node x_2 in φ_1; and u_2 in Figure 1 matches r_1[Institution] *w.r.t.* node p_1 in φ_1. Moreover, the relationship from u_1 to u_8 is bornOnDate and from u_1 to p_1 is worksAt. That is, both value constraints and structural constraints enforced by φ_1 are satisfied. Consequently, we can conclude that r_1[Name, DOB, Institution] are correct.

(2) Proof Negative. Consider rule φ_2. We can find that u_1 in



(a) Rule φ_1 (b) Rule φ_2 (c) Rule φ_3 (d) Rule φ_4

Fig. 4. Sample detective rules

Figure 1 matches r_1[Name] *w.r.t.* node w_1 in φ_2; u_2 in Figure 1 matches r_1[Institution] *w.r.t.* node w_2 in φ_2; u_5 in Figure 1 matches r_1[City] *w.r.t.* node n_2 in φ_2. Moreover, it can find a node u_7, with value *Haifa*, in Figure 1 that satisfies the constraints imposed on p_2 in φ_2. Combined with the other edge relationships from u_1 to u_2, and u_1 to u_5, we can confirm that r_1[City] $=$ *Karcag* is an error.

(3) Correction. Following case (2), we know r_1[City] should be repaired to *Haifa*.

Other rules will be discussed later in this paper. □

Remark. There might exist multiple repairs (*i.e.,* multi-version ground truth) for one error, and each makes sense, *e.g.,* one country may have multiple capitals and one person may have different nationalities. For the simplicity of the discussion, we assume that there is only one repair for this moment, *i.e.,* the corresponding relationship in the KB is functional. We will present algorithms to handle multiple repairs in latter sections. Also, we allow only one negative node n in the rule, which is also to simplify our discussion. It is straightforward to extend from one negative node (*i.e.,* one relationship) to a negative path (*i.e.,* a sequence of nodes) in order to identify an error.

III. Rule Generation and Applications

We first discuss how to generate DRs (Section III-A). We then describe the semantics of applying multiple DRs (Section III-B). We will also study the consistency problem of a set of DRs (Section III-C).

A. Generating Detective Rules by Positive/Negative Examples

Generating DRs is not easy. Obviously, the user has to be involved to identify the validity of DRs before they could be applied. Our aim is to make the user's life easier, by automatically computing a set of DRs to be verified.

We propose to *generate rules by examples*. Let D be a table of relation R and K a KB. We only discuss how to generate rules for one attribute $A \in R$, and the rules for the other attributes can be generated similarly. Let P be a set of positive tuple examples, *i.e.,* all values are correct. Let N be a set of negative examples, where only A-attribute values are wrong.

Algorithm DR Generation. We describe our algorithm below.

S1. [Schema-level matching graphs for P.] We use existing solutions [7] to compute a set $\mathcal{G}^+$ of schema-level matching graphs using the KB K, for the positive examples P. These correspond to the positive semantics of the table.

In a nutshell, given a set of correct tuples, the algorithm will map tuple values to KB instances to find their types and relationships. For instance, if we give two tuples with correct values as t_1*(China, Beijing)* and t_2*(Japan, Tokyo)*. The algorithm can find out that the first (resp. second) column has type country (resp. city) and their relationship is country hasCaptial city.

S2. [Schema-level matching graphs for N.] Similarly, we compute a set $\mathcal{G}^-$ of schema-level matching graphs, for the negative examples N. These correspond to negative semantics that errors in attribute A might have.

For instance, if we give two tuples with wrong values as t_1(*China, Shanghai*) and t_2 (*Japan, Kyoto*). The algorithm can find out the first (resp. second) column has type country (resp. city) and their relationship is city locatedIn country.

S3. [Candidate DR Generation.] For each graph $\mathbf{G}_i \in \mathcal{G}^+$ and each graph $\mathbf{G}_j \in \mathcal{G}^-$, if $\mathbf{G}_i$ and $\mathbf{G}_j$ have only one different node, that is, $u_i \in \mathbf{G}_i$ and $u_j \in \mathbf{G}_j$ are different and the two graphs $\mathbf{G}_i \backslash \{u_i\}$ and $\mathbf{G}_j \backslash \{u_j\}$ are isomorphic. We merge $\mathbf{G}_i$ and $\mathbf{G}_j$ as one DR, where u_i (resp. u_j) becomes the positive (resp. negative) node of the generated DR.

One can merge the outputs from **S1** and **S2**, the positive semantics country hasCaptial city and the negative semantics city locatedIn country, to generate one DR.

The above process will generate a set of candidate DRs. Indeed, the number is not large so the user can manually pick. Our claim is that, compared with asking the user to do the eyeballing exercise to write DRs manually, the above algorithm is simple but useful in practice.

B. Applying Multiple Detective Rules

We will start by discussing how to apply one DR, followed by using multiples DRs. For simplicity, at the moment, we assume that each DR will return a single repair.

Applying One Rule. Consider one DR $\varphi : \mathbf{G}(\mathbf{V}, \mathbf{E})$, a tuple t, and a KB K. Applying φ to t has only two cases: (1) *Proof positive:* the attribute values $t[\mathsf{col}(\mathbf{V_e} \cup \{p\})]$ are correct; (2) *Proof negative and correction:* the attribute values $t[\mathsf{col}(\mathbf{V_e})]$ are correct, the attribute value $t[\mathsf{col}(n)]$ is wrong, we will update $t[\mathsf{col}(n)]$ using an instance x drawn from the KB K.

We use the symbol "+" to mark a value as positive, which is confirmed either from the above (1), the evidence attributes $\mathbf{V_e}$ from the above (2), or a wrong value $t[\mathsf{col}(n)]$ as in the above (2) but has been corrected and thus is marked as positive. The other attributes that are not marked as positive are those whose correctness is *unknown*.

Marked Tuples. A tuple is a *marked tuple* if at least one of its attribute values has been marked as positive.

Example 6: [Apply one rule.] We discuss how to apply rule φ_2 to tuple r_1 by following Example 5 cases (2) & (3). $r_1[\mathsf{Name}]$ and $r_1[\mathsf{Institution}]$ are identified to be correct and $r_1[\mathsf{City}]$ is wrong. After changing $r_1[\mathsf{City}]$ to *Haifa*, it will mark r_1 as r_1'(*Avram Hershko*$^+$, *1937-12-31, Israel, Albert Lasher Award for Medicine, Israel Institute of Technology*$^+$, *Haifa*$^+$). □

Applying Multiple Rules. When applying multiple DRs, we need to make sure that the values that have been marked as positive cannot be changed any more by other DRs. Consider a *marked tuple* t of relation R that the attributes X have been marked as positive. We say that a DR φ is *applicable* to t, if (i) it will not change $t[X]$; and (ii) it can mark some values in $t[R\backslash X]$ as positive, with (*i.e.,* proof negative and correction) or without (*i.e.,* proof positive) value updates.

Fixpoint. A *fixpoint* of applying a set of DRs on a tuple t is the state that no more rules are further applicable.

Example 7: After applying rule φ_2 to tuple r_1 as in Example 6, rule φ_3 is applicable. Applying rule φ_3 will not change any value, but will mark the tuple as r_1''(*Avram Hershko*$^+$, *1937-12-31, Israel*$^+$, *Albert Lasher Award for Medicine, Israel Institute of Technology*$^+$, *Haifa*$^+$). Rule φ_4 will repair $r_1''[\mathsf{Prize}]$ and rule φ_1 marks $r_1''[\mathsf{DOB}]$ as positive. At the end, tuple r_1 is modified to r_1'''(*Avram Hershko*$^+$, *1937-12-31*$^+$, *Israel*$^+$, *Nobel Prize in Chemistry*$^+$, *Israel Institute of Technology*$^+$, *Haifa*$^+$). It is a fixpoint, since no more rule can be further applied. □

Termination Problem. It asks whether applying a set of DRs for any tuple t, whether a fixpoint can be achieved. Note that when applying any rule to t, we have that the set of marked positive attributes will strictly increase. That is, up to $|R|$ DRs can be applied to any tuple and the termination is naturally assured.

Multiple-version Repairs. Although not desired, the case that there are multiple-version repairs for one error by using one DR does happen. Note that this is different from IC-based repair to guess which value is wrong in *e.g.,* (*Netherlands, Rotterdam*) for $\{\mathsf{country}, \mathsf{capital}\}$. Instead, in our case, we know that *Rotterdam* is not the capital of *Netherlands*, and we find two repairs, *Amsterdam* and *Den Hagg*, and both are correct. In reality, when the user picks DRs (Section III-A), they will pick the ones that semantically, the repair is approximately functional, *e.g.,* the capital of a country or a nationality of a person, not a city of a country or a hobby of a person.

When multiple-version repairs happen for applying one DR to a tuple t, instead of having one marked tuple t', we generate multiple marked tuples T. These tuples T mark exactly the same set of attributes as positive, and these tuples are different only on one attribute *w.r.t.* the negative node in the given DR. This can be easily propagated to apply multiple DRs. Naturally, the *termination* analysis holds also for multiple-version repairs.

C. Consistency Analysis

An important problem of applying any data cleaning rules is to make sure that it makes sense to put these rules together.

Consistency Problem. Let Σ be a set of DRs and K a KB. Σ is said to be *consistent w.r.t.* K, if given any tuple t, all the possible repairs via Σ and K terminate in the same fixpoint(s), *i.e.,* the repair is unique.

Theorem 1: *The consistency problem for detective rules is coNP-complete, even when the knowledge base K is given.* □

We assume the values of a subset of attributes are initially given. Hence, the question is to ask whether there exists a tuple which has the same values in the subset of attributes and have more than one fixpoint.

Proof sketch: One can adapt the proof of Theorem 2 in [20] for our problem. The coNP upper bound can be proved by building a procedure to decide whether two rules are not satisfiable (*i.e.,* a "no" instance) – and therefore detecting consistency is coNP. The coNP-hard lower bound can be proved by reducing its complement problem to the 3SAT problem which is known to be NP-complete. □

The above consistency problem is hard, since one has to guess all the tuples that the given rules are applicable in an arbitrary order. Oftentimes, in practice, we only care about whether the rules are consistent *w.r.t.* a specific dataset D. Fortunately, the problem of checking the consistency when D is present becomes PTIME.

Corollary 2: *Given a relation D and a* KB K, i.e., $|R|$ *is a constant, the consistency problem is PTIME.* □

Given each tuple t over relation R, a set Σ of DRs, and a KB K, it has up to $|\Sigma|^{|R|}$ orders of applying the rules, where $|R|$ is a constant. Also, the number of tuples is $|D|$. Naturally, we can check whether Σ is consistent for D in PTIME.

The above result brings us to the bright side that practically it is feasible to make sure that a set of DRs is consistent for the dataset at hand. In our experiments, when a set of rules is selected, we run them on random sample tuples to check whether they always compute the same results. If not, we will ask users to double check the selected rules. In the following of the paper, we build our discussion using consistent DRs.

IV. Detective in Action

Given a set of consistent DRs, we first present a basic repair algorithm (Section IV-A), followed by an optimized algorithm to speed-up the repairing process (Section IV-B). Finally, we extend our methods to support multiple repairs (Section IV-C).

A. Basic Repairing

When a set Σ of DRs is consistent, for any tuple t, applying Σ to t will get a unique final result, which is also known as the Church-Rosser property [1]. Hence, the basic solution is

```
Algorithm 1: Basic Repair Algorithm
Input:  a tuple $t$, a set $\Sigma$ of consistent DRs.
Output: a repaired tuple $t$.
1. POS $\leftarrow \emptyset$;
2. while $\exists\varphi : \mathbf{G}(\mathbf{V_e} \cup \{p, n\}, \mathbf{E}) \in \Sigma$ that is applicable to $t$ do
3.     if $\exists$KB instances match $\mathbf{V_e} \cup \{p\}$ with $t[\mathsf{col}(\mathbf{V_e} \cup \{p\})]$ then
4.         POS $\leftarrow$ POS $\cup\ \mathsf{col}(\mathbf{V_e} \cup \{p\})$;        // Proof positive
5.     else if $\exists$KB instances $I \cup \{x_n\}$ that match $\mathbf{V_e} \cup \{n\}$ with
           $t[\mathsf{col}(\mathbf{V_e} \cup \{n\})]$ and $\exists$ a KB instance $x_p$ such that
           $I \cup \{x_p\}$ match $\mathbf{V_e} \cup \{p\}$ and $x_p \neq x_n$ then
6.         $t[\mathsf{col}(n)] \leftarrow x_p$;        // Proof negative and correction
7.         POS $\leftarrow$ POS $\cup\ \mathsf{col}(\mathbf{V_e} \cup \{p\})$ ;
8.     $\Sigma \leftarrow \Sigma \backslash \{\varphi\}$;
9. return $t$;
```

a chase-based process to iteratively pick a rule that can be applied until a fixpoint is reached, *i.e.,* no rule can be applied.

Algorithm 1. It uses a set to keep track of the attributes marked to be positive in t, initialized as empty (line 1). It picks one rule that is applicable to t in each iteration until no DR can be applied (lines 2-7). In each iteration, if there exist KB instances that match the nodes $\mathbf{V_e} \cup \{p\}$ with $t[\mathsf{col}(\mathbf{V_e} \cup \{p\})]$, the attributes $\mathsf{col}(\mathbf{V_e} \cup \{p\})$ are marked as positive (line 3-4). Otherwise, (i) if there exist KB instances $I \cup \{x_n\}$, such that they match nodes $\mathbf{V_e} \cup \{n\}$ with $t[\mathsf{col}(\mathbf{V_e} \cup \{n\})]$; and (ii) if there also exist KB instances $I \cup \{x_p\}$ that if we update $t[\mathsf{col}(n)]$ to x_p as t', $I \cup \{x_p\}$ match nodes $\mathbf{V_e} \cup \{p\}$ with $t'[\mathsf{col}(\mathbf{V_e} \cup \{p\})]$, and (iii) if $x_p \neq x_n$, it will repair this error by the value x_p and mark it as positive (lines 5-7). Afterwards, the rule will be removed from Σ since each rule can be applied only once (line 8). Finally, a repaired tuple is returned (line 9).

Complexity. The loop (lines 2-8) iterates at most $|\Sigma|$ times. In each iteration, it at most checks $|\Sigma|$ unused rules to find an applicable one. Within each loop, the worse case of checking each rule node $u \in \mathbf{V}$ is $O(|C||X|)$ where $|C|$ is the number of instances belonging to $\mathsf{type}(u)$ and $O(|X|)$ is the complexity of calculating the similarity between $t[\mathsf{col}(u)]$ and a KB instance. When checking whether $t[\mathsf{col}(u)]$ and $t[\mathsf{col}(v)]$ have the relationship $\mathsf{rel}(e)$ for each edge $e : (u, v) \in \mathbf{E}$ or drawing the correct value from KB needs $O(1)$ by utilizing a hash table. Thus, the algorithm runs in $O(|\Sigma|^2 \times (|C||X||\mathbf{V}| + |\mathbf{E}|))$ time, where $|\mathbf{V}|$ is the number of nodes and $|\mathbf{E}|$ is the number of edges in the rule.

B. Fast Repairing

We improve the above algorithm from three aspects.

(1) Rule Order Selection. Note that in Algorithm 1, when picking a rule to apply after the tuple has been changed, in the worst case, we need to scan all rules, even if some rules have been checked before. Naturally, we want to avoid checking rules repeatedly in each iteration.

The observation is that, applying a rule φ will affect another rule φ' only if φ changes some tuple value that φ' needs to check. More concretely, consider two DRs $\varphi : \mathbf{G}(\mathbf{V}, \mathbf{E})$ where $\mathbf{V} = \mathbf{V_e} \cup \{p, n\}$, and $\varphi' : \mathbf{G}'(\mathbf{V}', \mathbf{E}')$ where $\mathbf{V}' = \mathbf{V}'_\mathbf{e} \cup \{p', n'\}$. If $\mathsf{col}(n) \in \mathsf{col}(\mathbf{V}'_\mathbf{e})$, *i.e.,* the first rule will change some value of the tuple that can be used as the evidence for the second rule, then φ should be applied before φ'.

Rule Graph. Based on the above observation, we build a *rule graph* $\mathbf{G^r}(\mathbf{V^r}, \mathbf{E^r})$ for a set Σ of DRs. Each vertex $v_r \in \mathbf{V^r}$ corresponds to a rule in Σ. There is an edge from rule φ to φ' if $\mathsf{col}(p) \in \mathsf{col}(\mathbf{V}'_\mathbf{e})$. Note that a circle may exist, *i.e.,* there might also have an edge from φ' to φ if $\mathsf{col}(p') \in \mathsf{col}(\mathbf{V_e})$.

When repairing a tuple, we follow the topological ordering of the rule graph to check the availability of rules. Note that, if a circle exists in the rule graph, it is hard to ensure that the rules in the circle can be checked only once. We first treat the circle as a single node $\widetilde{v}$ to get the global order of Σ. When checking node $\widetilde{v}$, we first find a rule φ in this circle that can be applied. Then the edges pointing to φ can be removed.

Example 8: Consider rules in Figure 4. There are two connected components $\{\varphi_1, \varphi_2, \varphi_3\}$ and $\{\varphi_4\}$. The first three rules should be checked in the order $\langle\varphi_1, \varphi_2, \varphi_3\rangle$, since φ_1 may change attribute Institution that belongs to the evidence nodes of φ_2, and in turn φ_2 may change attribute City that is in the evidence nodes of φ_3. Checking φ_4 is irrelevant of the other rules, and thus it only needs to be checked once. □

(2) Efficient Instance Matching. The node in DR provides a similarity function to map values between schema R and KB K. If it is "=" (the cell $t[\mathsf{col}(u)]$ must be equal to an instance with $\mathsf{type}(u)$ in KB), we can just find all instances with $\mathsf{type}(u)$ and use a hash table to check whether $t[\mathsf{col}(u)]$ matches one of them. Otherwise, it is time-consuming to calculate the similarity between $t[\mathsf{col}(u)]$ and each instance. To improve the similarity-based matching, we use a signature-based framework [21]. For each $\mathsf{type}(u)$, we generate signatures for each instance in KB belonging to $\mathsf{type}(u)$. If a cell value in a column $\mathsf{col}(u)$ can match an instance (the similarity is larger than a threshold), they must share a common signature. In other words, for each cell value, we only need to find the instances that share common signatures with the cell. To this end, we build a signature-based inverted index. For each signature, we maintain an inverted list of instances that contain the signature. Given a cell value, the instances on the inverted list of signatures which also belong to the cell value are similarity-based matching candidates. In this way, we do not need to enumerate every instance.

(3) Sharing Computations on Common Nodes Between Different Rules. Note that a node can be used in multiple rules and it is expensive to check the node for every rule. To address this issue, we want to check each node only once. Furthermore, we want to build indexes to quickly check that, after a tuple has been updated, which rules are possibly affected. After a rule φ is applied, it marks attributes $\mathsf{col}(\mathbf{V_e} \cup \{p\})$ as positive. The attributes that are marked as positive cannot be changed by any other rules. Hence, after a tuple is updated by φ, all the rules φ' satisfying $\mathsf{col}(p') \in \mathsf{col}(\mathbf{V_e} \cup \{p\})$ can be safely removed. We utilize inverted lists to track these useless rules and in the meantime, to avoid repeated calculations that are shared between different rules. Similarly we can avoid checking the same relationship in different rules multiple times.

To achieve this, we propose a novel inverted list that can be used interchangeably for both nodes and relationships.

Inverted Lists. Each *inverted list* is a mapping from a key to

```
Algorithm 2: Fast Repair Algorithm
Input:   a tuple $t$, a set $\Sigma$ of consistent DRs, inverted lists $\mathcal{I}$.
Output:  a repaired tuple $t$.
1.  sort $\Sigma$ in topological ordering;
2.  for each $\varphi \in \Sigma$ do
3.    for each $vertex\ or\ edge\ o \in \varphi : \mathbf{G}(\mathbf{V}, \mathbf{E})$ do
4.      if $t$ matches $o$ then
5.        for each $(\varphi', o')\ in\ \mathcal{I}(o)$ do
6.          mark the vertex or edge $o' \in \varphi'$ as already checked;
7.      else
8.        for each $(\varphi', o')\ in\ \mathcal{I}(o)$ do
9.            $\Sigma \leftarrow \Sigma \backslash \{\varphi'\}$;
10. if $\varphi$ is applicable to $t$ then
11.   update or mark $t$ by $\varphi$;
12.   if $t[\mathsf{col}(p)]$ is marked as positive then
13.     delete rules in $\Sigma$ that also update $\mathsf{col}(p)$;
14.     for each $o \in \{p\} \cup \{e|edge\ e\ connected\ p\}$ do
15.         for each $(\varphi', o')\ in\ \mathcal{I}(o)$ do
16.           mark the vertex or edge $o' \in \varphi'$ as checked;
17. else $\Sigma \leftarrow \Sigma \backslash \{\varphi\}$;
18. return $t$;
```

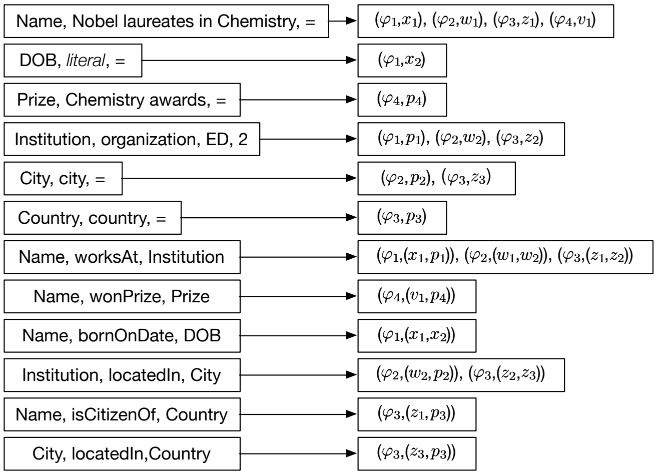
a set Ψ of values. Each key is (i) a match between a column in R and a class in KB with similarity function $\approx_u$ or (ii) a relationship/property in KB that describes the relationship between two columns. Each value in Ψ is a pair (φ, o) where $\varphi \in \Sigma$ and o is either a vertex or an edge in $\mathbf{G} \backslash \{n\}$. Each pair in Ψ satisfies that the vertex (or edge) o must contain the node (or relationship) in the key. The inverted lists *w.r.t.* rules in Figure 4 are shown in Figure 5.

We are now ready to present the fast repair algorithm.

Algorithm 2. It first sorts the rules in Σ in topological ordering (line 1) and then checks the rules in turn (lines 2-17). For each rule, every vertex and edge in a DR $\varphi : \mathbf{G}(\mathbf{V}, \mathbf{E})$ will be visited. If it has not been checked, we detect whether $t[\mathsf{col}(u)]$ belongs to $\mathsf{type}(u)$ for vertex u or whether $t[\mathsf{col}(u)], t[\mathsf{col}(v)]$ have relationship $\mathsf{rel}(e)$ for edge e (lines 3-9). If so, we mark this vertex or edge in other rule φ' as already being checked using $\mathcal{I}$ (lines 4-6). Otherwise, we delete φ' from Σ (lines 8-9). We apply rule φ to tuple t if it is applicable (lines 10-16). Otherwise, rule φ is deleted (line 17). If $t[\mathsf{col}(p)]$ is marked as positive, we delete all rules which also update $\mathsf{col}(p)$ from Σ (line 13). Meanwhile, for each rule φ' that also contains p (as evidence node) or the edge connected to p, it also should be marked as already being checked (lines 14-16).

Complexity. The complexity of sorting rules is $O(|\Sigma| + |\mathbf{E}^{\mathbf{r}}|)$ where $\mathbf{E}^{\mathbf{r}}$ is the number of edges in the rule graph. Note that we only need to sort once and apply to all tuples. It is obvious that the outer loop (lines 2-17) runs at most $|\Sigma|$ times. For each DR $\varphi : \mathbf{G}(\mathbf{V}, \mathbf{E})$, the worst case of checking each vertex and edge needs $O(|C||X||\mathbf{V}| + |\mathbf{E}|)$ as stated above even utilizing the similarity indexes. Thus, the algorithm requires $O(|\Sigma| \times (|C||X||\mathbf{V}| + |\mathbf{E}|))$.

Example 9: Consider tuple r_3 in Table I, four DRs in Figure 4 and the invert lists in Figure 5. The rules will be checked in the order $\langle \varphi_4, \varphi_1, \varphi_2, \varphi_3 \rangle$. Let $\mathsf{checked}_\varphi$ denotes the set storing the vertexes and edges in φ that have been checked to be matched with tuple t.



Name, Nobel laureates in Chemistry, =	(φ_1,x_1), (φ_2,w_1), (φ_3,z_1), (φ_4,v_1)
DOB, *literal*, =	(φ_1,x_2)
Prize, Chemistry awards, =	(φ_4,p_4)
Institution, organization, ED, 2	(φ_1,p_1), (φ_2,w_2), (φ_3,z_2)
City, city, =	(φ_2,p_2), (φ_3,z_3)
Country, country, =	(φ_3,p_3)
Name, worksAt, Institution	$(\varphi_1,(x_1,p_1))$, $(\varphi_2,(w_1,w_2))$, $(\varphi_3,(z_1,z_2))$
Name, wonPrize, Prize	$(\varphi_4,(v_1,p_4))$
Name, bornOnDate, DOB	$(\varphi_1,(x_1,x_2))$
Institution, locatedIn, City	$(\varphi_2,(w_2,p_2))$, $(\varphi_3,(z_2,z_3))$
Name, isCitizenOf, Country	$(\varphi_3,(z_1,p_3))$
City, locatedIn,Country	$(\varphi_3,(z_3,p_3))$

Fig. 5. Rule Indexes

For rule φ_4, $r_3[\mathsf{Name}]$ can match node v_1. We maintain $\mathsf{checked}_{\varphi_4} = \{v_1\}$. Besides, by utilizing the inverted lists, we have $\mathsf{checked}_{\varphi_1} = \{x_1\}$, $\mathsf{checked}_{\varphi_2} = \{w_1\}$ and $\mathsf{checked}_{\varphi_3} = \{z_1\}$. The negative node n_4 can also be matched, thus $r_3[\mathsf{Prize}]$ will be updated to *Nobel Prize in Chemistry*. Tuple r_3 becomes r'_3(*Roald Hoffmann*$^+$*, 1937-07-18, Ukraine, Nobel Prize in Chemistry*$^+$*, Cornell University, Ithaca*).

Then for rule φ_1, since r'_3 can match nodes x_1, x_2, p_1 and edges (x_1, x_2),(x_1, p_1), we have r''_3(*Roald Hoffmann*$^+$*, 1937-07-18*$^+$*, Ukraine, Nobel Prize in Chemistry*$^+$*, Cornell University*$^+$*, Ithaca*). We can expand $\mathsf{checked}_{\varphi_2} = \{w_1, w_2, (w_1, w_2)\}$ and $\mathsf{checked}_{\varphi_3} = \{z_1, z_2, (z_1, z_2)\}$ by utilizing the inverted lists.

When considering rule φ_2, we only need to check the negative and positive nodes based on $\mathsf{checked}_{\varphi_2}$. As $r''_3[\mathsf{Institution}]$ is actually located in $r''_3[\mathsf{City}]$. We have r'''_3(*Roald Hoffmann*$^+$*, 1937-07-18*$^+$*, Ukraine, Nobel Prize in Chemistry*$^+$*, Cornell University*$^+$*, Ithaca*$^+$). Meanwhile, $\mathsf{checked}_{\varphi_3} = \{z_1, z_2, z_3, (z_1, z_2), (z_2, z_3)\}$

Based on $\mathsf{checked}_{\varphi_3}$, we only need to examine the negative and positive nodes. Since tuple r'''_3 matches node n_3 and the relationship (z_1, n_3), $r'''_3[\mathsf{Country}]$ will be updated to *United States* by applying φ_3. Tuple r''''_3(*Roald Hoffmann*$^+$*, 1937-07-18*$^+$*, United States*$^+$*, Nobel Prize in Chemistry*$^+$*, Cornell University*$^+$*, Ithaca*$^+$) is a fixpoint. □

C. Multiple-version Repairs

We can extend our methods to support multiple-version repairs. The basic idea is, when multiple versions are identified to repair one tuple, instead of having only one updated tuple, we have multiple updated tuples. The workflow is the same as the fast repair algorithm. Below we illustrate by an example.

Example 10: Consider tuple r_4 in Table I, four DRs in Figure 4. The rules will be checked in the order $\langle \varphi_4, \varphi_1, \varphi_2, \varphi_3 \rangle$. We maintain a set T to keep track of the intermediate repairs. At first, $T = \{r_4\}$.

Rule φ_4 will not repair any value but mark $r_4[\mathsf{Name}]$ and $r_4[\mathsf{Prize}]$ as positive. As for rule φ_1, tuple r_4 matches nodes x_1, x_2, n_1 and edges $(x_1, x_2), (x_1, n_1)$, so φ_1 can be applied.

From the KB, we know that *Melvin Calvin* worked in two institutions $\{$*University of Manchester, UC Berkeley*$\}$. Thus, r_4 can be repaired in two ways: r'_4*(Melvin Calvin*$^+$*, 1911-04-08*$^+$*, United States, Nobel Prize in Chemistry*$^+$*, University of Manchester*$^+$*, St. Paul)* and r''_4*(Melvin Calvin*$^+$*, 1911-04-08*$^+$*, United States, Nobel Prize in Chemistry*$^+$*, UC Berkeley*$^+$*, St. Paul)*. r'_4 is added into T temporarily and r''_4 continues to be repaired.

Rule φ_2 will update r''_4[City] to *Berkeley* from *St. Paul* and φ_3 marks r''_4[Country] as positive. r'''_4*(Melvin Calvin*$^+$*, 1911-04-08*$^+$*, United States*$^+$*, Nobel Prize in Chemistry*$^+$*, UC Berkeley*$^+$*, Berkeley*$^+$*)* is a fixpoint. At this moment, we get one result r'''_4 and $T = \{r'_4\}$.

Then r'_4 will be taken out from T for further repair and rule φ_2, φ_3 are left to be checked. Rule φ_2 will update r'_4[City] to *Manchester* from *St. Paul* and φ_3 marks r'_4[Country] as positive. r''''_4*(Melvin Calvin*$^+$*, 1911-04-08*$^+$*, United States*$^+$*, Nobel Prize in Chemistry*$^+$*, University of Manchester*$^+$*, Manchester*$^+$*)* is another fixpoint. Now we have two valid repair r'''_4 and r''''_4, and $T = \emptyset$. □

V. Experimental Study

A. Experimental Setup

Datasets. We used a set of small Web tables, a real-world dataset and a synthetic dataset. Note that we used the datasets that are covered by general purpose KBs.

(1) WebTables. This dataset contains 37 tables[2], with the average number of tuples 44. We chose it since they are representative to cover a wide range of other general purpose Web tables.

(2) Nobel. It contains 1069 tuples about Nobel laureates, obtained by joining two tables from Wikipedia: List of Nobel laureates by country[3] and List of countries by Nobel laureates per capita[4]. We tested this case to see how our approach performs for personal information, an important topic considered in many applications.

(3) UIS. It is a synthetic dataset generated by the UIS Database Generator[5]. We generated 100K tuples.

Knowledge Bases. We used Yago [19] and DBpedia [23] for our experiments. It is known that both Yago and DBpedia share general information of generating a structured ontology. However, the difference is that Yago focuses more on the taxonomic structure, *e.g.,* richer type/relationship hierarchies. This indeed makes the experiment more interesting to see how taxonomic structure plays the role for mapping the information between relations and KBs. The number of aligned classes and relations of testing datasets are given in Table II.

Noise. We did not inject noise to WebTables because they are dirty originally. Noises injected to Nobel and UIS have two types: (i) typos; (ii) semantic errors: the value is replaced with a different one from a semantically related attribute. Errors were produced by adding noises with a certain rate e%, *i.e.,* the percentage of dirty cells over all data cells.

	Yago		DBPedia	
	#-class	#-relationship	#-class	#-relationship
WebTables	42	30	51	30
Nobel	5	4	5	4
UIS	5	5	5	4

TABLE II. Datasets (aligned classes and relations)

WebTables		Precision	Recall	F-measure	#-POS
DRs	Yago	1	0.38	0.55	1469
	DBpedia	1	0.43	0.60	1326
KATARA	Yago	0.73	0.40	0.52	864
	DBpedia	0.78	0.46	0.58	752
Nobel		Precision	Recall	F-measure	#-POS
DRs	Yago	1	0.70	0.82	1543
	DBpedia	1	0.54	0.70	715
KATARA	Yago	0.74	0.68	0.71	396
	DBpedia	0.64	0.49	0.56	189
UIS		Precision	Recall	F-measure	#-POS
DRs	Yago	1	0.73	0.84	77001
	DBpedia	1	0.63	0.77	57703
KATARA	Yago	0.67	0.77	0.72	35084
	DBpedia	0.63	0.57	0.60	25152

TABLE III. Data Annotation and Repair Accuracy

Detective Rules. The DRs were generated as described in Section III-A, verified by experts. For WebTables, we totally generated 50 DRs, and for Nobel and UIS, we generated 5 DRs for each table. There are few cases that multi-version repairs appear in our experiments. In this case, if one of the repairs matches the ground truth value, we treat it as a correct repair.

Algorithms. We implemented the following algorithms: (i) bRepair: the basic repair algorithm (Section IV-A); and (ii) fRepair: the fast repair algorithm (Section IV-B). For comparison, we have implemented KATARA [7], which is also a KB-based data cleaning system. We also compared with two IC-based repairing algorithms: Llunatic [17] and constant CFDs [14].

Measuring Quality. We used precision, recall and F-measure to evaluate the repairing quality: precision is the ratio of correctly repaired attribute values to the number of all the repaired attributes; and recall is the ratio of correctly repaired attribute values to the number of all erroneous values; and F-measure is the harmonic mean of precision and recall. We manually repaired WebTables and regarded them as ground truth. Besides, knowledge bases cannot cover the whole tables. For the other tables, we mainly evaluated the tuples whose value in *key* attribute (*e.g.,* Name *w.r.t.* Nobel or State *w.r.t.* UIS) have corresponding entities in KBs.

Experimental Environment. All methods were written in Java and all tests were conducted on a PC with a 2.40GHz Intel CPU and 64GB RAM. For efficiency, each experiment was run six times, and the average results were reported.

B. Experimental Results

We tested DRs from three aspects. Exp-1: The comparison with other KB-based data cleaning methods. Exp-2: The comparison with IC-based cleaning on tables. Exp-3: Efficiency and scalability of our solutions.

[2] https://www.cse.iitb.ac.in/~sunita/wwt/

[3] https://en.wikipedia.org/wiki/List_of_Nobel_laureates_by_country

[4] https://en.wikipedia.org/wiki/List_of_countries_by_Nobel_laureates_per_capita

[5] http://sherlock.ics.uci.edu/data.html

Exp-1: Comparison with KB Powered Data Cleaning. We compared with KATARA [7], the most recent data cleaning system that is powered by KBs and experts in crowdsourcing. Although KATARA can mark data as correct, it cannot automatically detect or repair errors. In fact, KATARA relies on experts to manually detect and repair errors.

In order to have a fair comparison by removing the expert sourcing factor, we revised KATARA by simulating expert behavior as follows. When there was a full match of a tuple and the KB under the table pattern defined by KATARA, the whole tuple was marked as correct. When there was a partial match, we revised KATARA by marking the minimally unmatched attributes as wrong. For repairing, since KATARA also computes candidate repairs, as presented in [7], we picked the one from all candidates that minimizes the repair cost.

(A) Data Repair. We first compared with KATARA about data repairing accuracy, using the datasets reported in Table II. For Nobel and UIS, the error rate was 10%.

Precision. Table III shows the results of applying DRs and using KATARA for data repairing. DRs were carefully designed to ensure trusted repair. Hence, not surprisedly, the precision was always 1. This is always true if the DRs are correct, when being carefully selected by the users. KATARA, on the other hand, relies on experts to make decisions. Once experts are absent, KATARA itself cannot decide how to repair, which result in relatively low precision as reported in the table.

Recall. As shown in Table III, for WebTables, DRs had lower recall than KATARA. It is because some of WebTables have few number of attributes. This is not enough to support the modifications of DRs. For example, considering the schema is (Author, Book), when t[Author] and t[Book] do not satisfy the relationship wrote, it is hard to ensure that which attribute is wrong. So our methods would not repair this kind of tables, in a conservative way. For Nobel and UIS, the recall of our algorithms were higher than KATARA. The reason is that KATARA does not support fuzzy matching. In order to find proper modifications, at least one attribute must be correct.

F-measure. Our method had higher F-measure than KATARA, because our method had higher precision and similar recall. Taken the explanations above, it is easy to see that DRs had comparable F-measure with KATARA for WebTables, and better F-measure in Nobel and UIS, as shown in Table III.

(B) Data Annotation. KATARA can also mark correct data, when a given tuple can find a full match in the given KB, relative to the table pattern they used. Note that in their paper, KATARA can mark wrong data. However, each wrong value has to be manually verified by experts. For comparison, given a tuple and a KB, if a partial match is found by KATARA, one way is to heuristically mark the matched part as correct and the unmatched part as wrong. This will cause both false positives and false negatives. Hence, in order to have a relatively fair comparison, we favor KATARA by only checking the full matches that they mark as correct.

Table III gives the results of both DRs and KATARA in marking data, listed in the last column #-POS. The results show that, even by ignoring the ability of data repairing, DRs can automatically mark much more positive data than KATARA. These information is extremely important for both heuristic and probabilistic methods, since the main reason they make false positives and false negatives is that they cannot precisely guess which data is correct or wrong.

Exp-2: Comparison with IC-based Repair. In this group of experimental study, we compared with IC-based repairing algorithms. Llunatic [17] involves different kinds of ICs and different strategies to select preferred values. For Llunatic, we used FDs and chose its frequency cost-manager. *Metric* 0.5 was used to measure the repair quality (for each cell repaired to a variable, it was counted as a partially correct change). For constant CFDs, they were generated from ground truth. We simulated the user behavior by repairing the right hand side of a tuple t based on a constant CFD, if the left side values of t were the same as the values in the given constant CFD. In this case, constant CFDs will make mistakes if the tuple's left hand side values are wrong.

Also, since there is not much redundancy in the WebTables that IC rely on to find errors (or violations), we tested using only Nobel and UIS datasets. We first evaluated the accuracy of repair algorithm over different error rates. We then varied the percentage of error types in Nobel and UIS to get better insight into the strong and weak points of DRs, compared with other IC-based approaches.

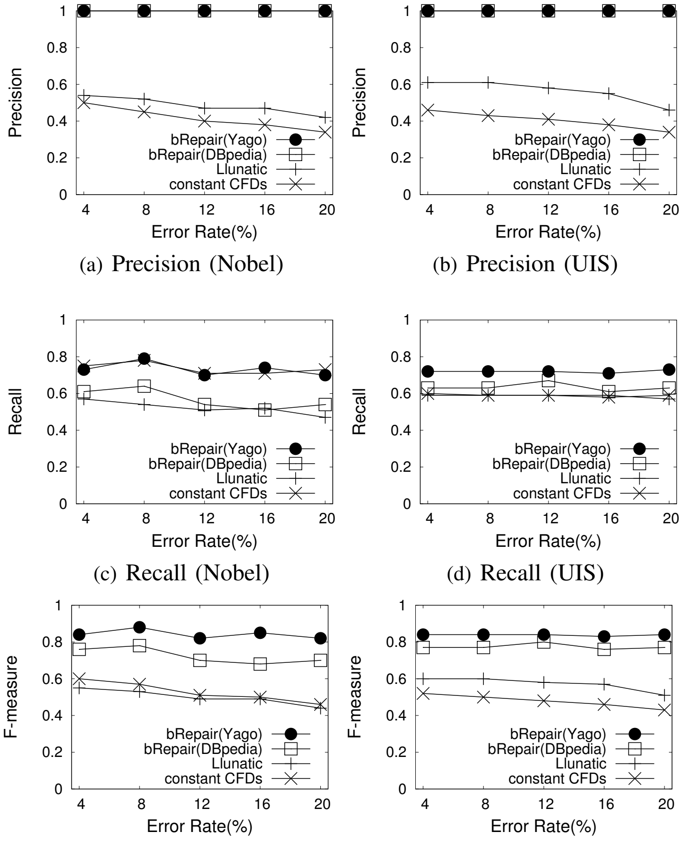
(A) Varying Error Rate. For Nobel and UIS, we studied the accuracy of our repair algorithm by varying the error rate from 4% to 20%, and reported the precision, recall and F-measure in Figure 6. The rates of different error types, *i.e.,* typos and semantic errors were equal, *i.e.,* 50-50.

We can see that our methods had stable performance when error rates increased. However, the precision and recall of Llunatic moderately decreased, since when more errors were injected, it became harder to detect errors and link relevant tuples for heuristic repair algorithm. The precision and recall of constant CFDs also decreased because there were more chances that errors happened on the left hand side of constant CFDs.

From Figure 6(d), we see that our algorithms did not have higher recall. This is because: (i) KBs cannot cover all attribute values in Nobel and UIS, *e.g.,* some City can find corresponding resource with property locatedIn to repair the attribute State but some cannot. Thus, some errors cannot be detected; (ii) to ensure the precision, we would not repair errors when the evidence was not sufficient; and (iii) if semantic errors were injected into the evidence nodes of DRs, we cannot detect and repair them. On the contrary, the other two methods would repair some potentially erroneous heuristically, which may increase their recall.

(B) Varying Typo Rate. We fixed the error rate at 10% and varied the percentages of typos from 0% to 100% (semantic errors from 100% to 0% corresponding) to evaluate the ability of capture typos and the semantic error. The experimental results are shown in Figure 7.

Figure 7 shows that Llunatic and our methods behaved better with typos than with semantic error. The reason is that they all chose to repair an error to the most similar candidate, which for typos is more likely to be correct value. On the contrary,

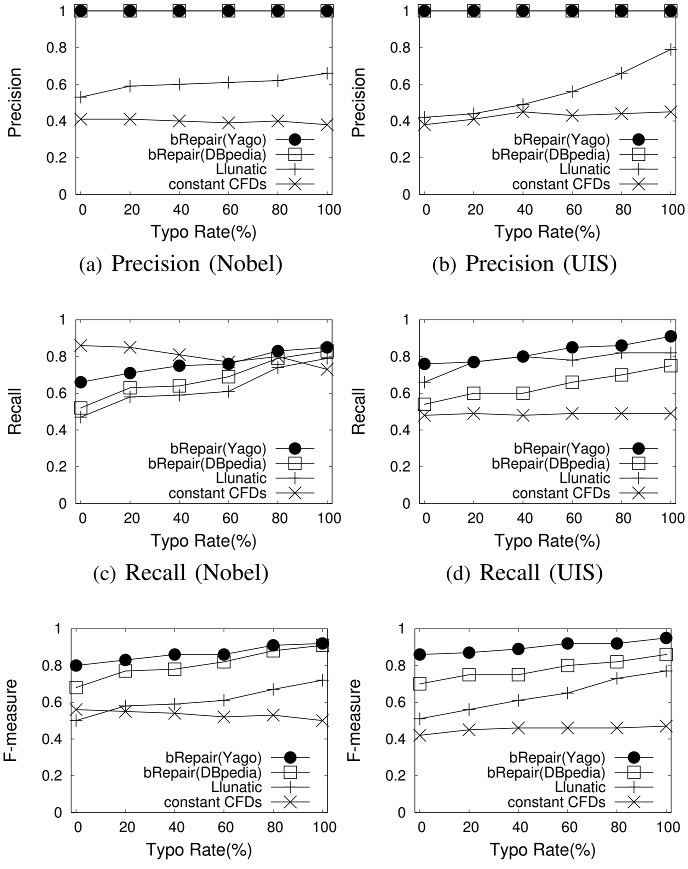


(a) Precision (Nobel) (b) Precision (UIS)

(c) Recall (Nobel) (d) Recall (UIS)

(e) F-measure (Nobel) (f) F-measure (UIS)

Fig. 6. Effectiveness (varying error rate)



(a) Precision (Nobel) (b) Precision (UIS)

(c) Recall (Nobel) (d) Recall (UIS)

(e) F-measure (Nobel) (f) F-measure (UIS)

Fig. 7. Effectiveness (varying typo rate)

if the semantic errors were added to the evidence nodes of DRs or left hand of FDs, none of us can detect that errors. Meanwhile, more $lluns$ (unknown defined in Llunatic) were introduced to make the table consistent. Constant CFDs do not support fuzzy matching. Thus, it is hard to say which type of errors it can detect better. These errors can be detected only when they are injected to the right hand of constraints.

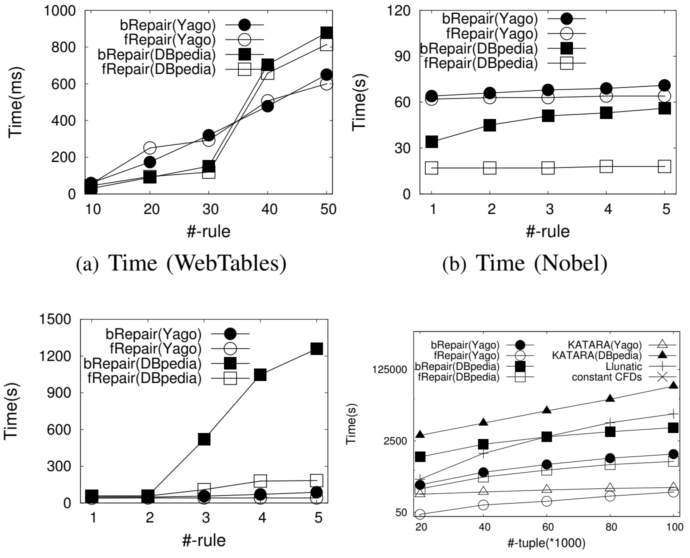
Exp-3: Efficiency Study. We evaluated the efficiency of our repair algorithms using WebTables, Nobel and UIS. We first varied the number of DRs to measure the performance of bRepair and fRepair. Then we studied the scalability of the algorithms utilizing the UIS Database Generator.

(A) Varying #-Rule. We varied the number of rules from 10 to 50 by a step of 10 for WebTables, and varied from 1 to 5 by a step of 1 for Nobel and UIS. The execution time were reported in Figure 8(a)-(c). The error rate of Nobel and UIS was fixed at 10% and we generated 20K tuples for UIS. To better represent the impact of indexes, we did not sum the time of reading and handling KBs.

There is no doubt that with the growing size of ruleset, fRepair was more efficient than bRepair. For example, when there were 5 DRs to repair UIS utilizing DBpedia, bRepair ran 1323s, while fRepair only ran 217s. For WebTables, fRepair was not so faster than bRepair. It is because the extra cost of sorting rules and keeping inverted lists became unnegligible when only a few of DRs were used for repairing a few tuples.

(B) Varying #-Tuple. In this part of experiment, we evaluated the scalability of our methods and compared with the other three repair algorithms: KATARA, Llunatic and constant CFDs. We utilized the UIS Database Generator and varied the number of tuples from 20K to 100K by a step of 20K, fixing the error rate at 10%. The experimental result was reported in Figure 8(d). Note that the time of reading and handling KBs was included in this part of experiments.

Figure 8(d) indicates that the impact of indexes became more and more remarkable with the growing the data size. For example, when there were 100K tuples to repair, the bRepair algorithm utilizing Yago ran 1216s, while fRepair only ran 152s. The fRepair algorithm always ran faster than Llunatic , and the time cost of Llunatic increased faster along with the number of tuples grew. The reason is that, Llunatic needed to consider multiple tuples to detect violations and holistically consider multiple violations to decide a repair strategy. Our methods also ran faster than KATARA especially for DBpedia, because KATARA needed to list all instance graphs and find the most similar one for each tuple. Note that constant CFDs use only instances, thus it can repair 100K tuples within 1s.



(a) Time (WebTables)

(b) Time (Nobel)

(c) Time (UIS)

(d) Time (UIS)

Fig. 8. Efficiency

Summary of Experimental Findings. (1) DRs are effective in using KBs for cleaning relations. Without experts being involved, DRs are more accurate than the state-of-the-art data cleaning system KATARA that also uses KBs (Exp-1). (2) DRs are more effective than IC-based data cleaning (Exp-2). Note that we did not compared with other rule-based algorithms that are also ensured correctness if the rules are correct. The only reason is that existing rule-based methods do not rely on KBs but on expert knowledge or master tables. (3) It is efficient and scalable to apply DRs (Exp-3), since repairing one tuple is irrelevant to any other tuple.

VI. Conclusion

In this paper, we have proposed detective rules. Given a relation and a KB, DRs tell us that which tuple values are correct, which tuple values are erroneous, and how to repair them if there is enough evidence in the KB, in a deterministic fashion. We have studied fundamental problems associated with DRs, such as consistency analysis. We have also proposed efficient data repairing algorithms using detective rules.

Acknowledgment

Guoliang Li was supported by 973 Program of China (2015CB358700), NSF of China (61422205, 61373024, 61632016, 61472198, 61661166012), Shenzhou, Tencent, FDCT/116/2013/A3, and MYRG105 (Y1-L3)-FST13-GZ.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. *TPLP*, 3(4-5), 2003.

[3] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, 18(1), 2009.

[4] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, 2005.

[5] F. Chiang and R. J. Miller. A unified model for data and constraint repair. In *ICDE*, 2011.

[6] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, 2013.

[7] X. Chu, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, N. Tang, and Y. Ye. KATARA: a data cleaning system powered by knowledge bases and crowdsourcing. In *SIGMOD*, 2015.

[8] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB*, 2007.

[9] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: a commodity data cleaning system. In *SIGMOD*, 2013.

[10] D. Deng, Y. Jiang, G. Li, J. Li, and C. Yu. Scalable column concept determination for web tables using large knowledge bases. *PVLDB*, 6(13), 2013.

[11] O. Deshpande, D. S. Lamba, M. Tourn, S. Das, S. Subramaniam, A. Rajaraman, V. Harinarayan, and A. Doan. Building, maintaining, and using knowledge bases: a report from the trenches. In *SIGMOD Conference*, 2013.

[12] W. Fan. Dependencies revisited for improving data quality. In *PODS*, 2008.

[13] W. Fan, Z. Fan, C. Tian, and X. L. Dong. Keys for graphs. *PVLDB*, 8(12), 2015.

[14] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 33(2), 2008.

[15] W. Fan, X. Jia, J. Li, and S. Ma. Reasoning about record matching rules. *PVLDB*, 2(1), 2009.

[16] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *VLDB J.*, 21(2), 2012.

[17] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC Data-Cleaning Framework. *PVLDB*, 2013.

[18] J. He, E. Veltri, D. Santoro, G. Li, G. Mecca, P. Papotti, and N. Tang. Interactive and deterministic data cleaning. In *SIGMOD*, 2016.

[19] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2: A spatially and temporally enhanced knowledge base from wikipedia. *Artif. Intell.*, 194, 2013.

[20] M. Interlandi and N. Tang. Proof positive and negative in data cleaning. In *ICDE*, 2015.

[21] G. Li, D. Deng, J. Wang, and J. Feng. PASS-JOIN: A partition-based method for similarity joins. *VLDB*, 5(3):253–264, 2011.

[22] G. Limaye, S. Sarawagi, and S. Chakrabarti. Annotating and searching web tables using entities, types and relationships. *PVLDB*, 2010.

[23] M. Morsey, J. Lehmann, S. Auer, and A. N. Ngomo. Dbpedia SPARQL benchmark - performance assessment with real queries on real data. In *ISWC*, 2011.

[24] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, 2001.

[25] S. Song, H. Cheng, J. X. Yu, and L. Chen. Repairing vertex labels under neighborhood constraints. *PVLDB*, 7(11), 2014.

[26] P. Venetis, A. Y. Halevy, J. Madhavan, M. Pasca, W. Shen, F. Wu, G. Miao, and C. Wu. Recovering semantics of tables on the web. *PVLDB*, 2011.

[27] M. Volkovs, F. Chiang, J. Szlichta, and R. J. Miller. Continuous data cleaning. In *ICDE*, 2014.

[28] J. Wang and N. Tang. Towards dependable data repairing with fixing rules. In *SIGMOD*, 2014.

[29] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 2011.